Table of Contents

- Forking QEMU to emulate and secure the Tillitis TKey
  - MC
  - Tillitis history
  - Tillitis TKey: a new kind of security token?
  - Tillitis Tkey: a small open source/open hardware computer
    - You can use the TKey for
    - Advantages
  - Basic TKey use with killer app: tkey-ssh-agent
    - But… It's a general computer!?
    - Measured boot
    - Advantages of measured boot
  - TKey specs
  - Hardware design and testing tools
  - FPGA chip
  - In the FPGA
  - Software
  - Status
  - The emulator
    - Why an emulator?
  - QEMU - the gold standard
    - Goal
    - Reality
  - Using the emulator
    - Starting a TKey emulator
    - Interacting with emulated TKey
    - Debugging with GDB and qemu
  - Staying up to date with upstream
  - Story time!
    - The bug

        - The exploit
        - Fix
        - Investigation
        - Order in memory
        - Order in memory on real hardware
        - But the signing context?
        - The real fix
    - Future of the emulator
    - Future of the TKey
    - Summary
    - The end

Forking QEMU to emulate and secure the Tillitis TKey

MC

- Michael “MC” Cardell Widerkrantz, mc@tillitis.se
- Member of Technical Staff @ https://tillitis.se/
- Personal: https://hack.org/~mc/ & gemini://gem.hack.org/mc/

Tillitis history

- Born at Mullvad VPN's Trustworthy Computing Research team.

TCR team split in September 2022:

- Glasklar Teknik AB https://www.glasklarteknik.se/ Projects:
  - https://system-transparency.org/
  - https://sigsum.org/
- Tillitis: The hardware department! https://tillitis.se/

Tillitis TKey: a new kind of security token?

- Joachim Strömbergson presented the TKey at foss-north 2023.
- “a radically open authentication platform that fits in your pocket”.

- “fully open - from circuit board to applications, and yet provides strong security foundations.”

Tillitis Tkey: a small open source/open hardware computer

- A RISC-V computer for sensitive computations.
- No secure enclave! No black box!
- Open source software and hardware (BSD2, CERN-OHL, some parts still GPLv2 but moving to BSD2.)
- https://github.com/tillitis
- https://dev.tillitis.se/

You can use the TKey for

- Authentication.
- Digital signatures.
- Hardware root of trust.
- (Signed) random number generator.
- Encryption.
- A protected environment for sensitive computations.
- Other things… It's a general computer!

Advantages

- The client (computer/mobile) decides the function of the TKey.
- No need for new hardware for new functionality.
- Can write custom software.
- No risk for persistent threats.
- Secrets and private keys are not stored persistently on the device.
- You can verify that the TKey comes from the vendor.
- You can make your own TKey, or just choose your own base hardware secret.

Basic TKey use with killer app: tkey-ssh-agent

I want to login to something, typically a server (or Github, Gitlab or even sign my Git commits):

- My client started the tkey-ssh-agent automatically when I logged in.
- I insert the TKey into my client.
- `ssh some-server`
- The agent automatically loads the device app `signer`, Ed25519 signatures.
- TKey starts to blink the status LED.
- I touch the sensor.
- “I'm in!”
- Just like any security token.

But… It's a general computer!?

- How can we trust general applications sent from the client?
- What if we don't share anything between the apps?
- …and guarantee software integrity?
- …by measuring the apps,
- and creating new secrets for this combination of app and device.
- These secrets never leave the TKey.

Measured boot

- Use immutable code to measure the application, mix in a hardware secret: get a new identity!
- Inspired by TCG DICE (nee RIoT from Microsoft Research): Trusted boot for constrained environments.

Advantages of measured boot

- Software integrity is guaranteed.
- (And verifiable, if you have a public key.)
- The measured identity can be used to create key material.
- Private keys are not stored on the TKey.
- Unlimited number of private keys.
- Secrets don't leak between device applications.

- Different than verified boot
  - **Forced** verified boot would lock TKey device apps to a specific vendor.
  - Unacceptable in an open platform!
- Compound Device Identity

  Result of measured boot:

  ```
  cdi := blake2s(uds, blake2s(application), uss)
  ```

  CDI is a cryptograpic mix of:

  - Unique Device Secret (UDS) in hardware, something the user **has**.
  - Optional User Supplied Secret (USS), something the user **knows**.
  - Measurement (hash digest) of TKey device application, the **integrity** of the application.

TKey specs

- 32 bit RISC-V PicoRV32 (Claire Wolf) softcore @ 18 MHz.
- 128 kiB RAM.
- Memory mapped hardware cores.
- Firmware mode/app mode.
- No interrupts.

- No persistent storage. (1 MiB flash usable during hardware dev.)
- No OS.

Hardware design and testing tools

- Only open source tools!
- Chip design in Verilog.
- Limits choice of FPGA chip.
- Limits choice of PCB manufacturers.
- Yosys & NextPNR for synthesis, place & route, mapping and timing.
- Icestorm tools for bitstream generation.
- Developed NVCM programming tools.
- Icarus and Verilator for module and systems simulation.
- PCB design with KiCAD.
- Everything published!

FPGA chip

- Lattice iCE40 UltraPlus UP5K FPGA.
- Good support in open source tools.
- Lockable internal configuration memory (NVCM).
- Limited resources (~5 k LUTs, 120 kbit block RAM, 1024 kbit SPRAM).
- Paying for reversing other FPGA chips.

In the FPGA

- CPU
- ROM
- RAM
- FW_RAM
- Timer
- UART
- UDS

- Touch sensor
- TRNG
- TK1
  - Security Monitor
  - GPIO (unused)

Software

- Emulator: friendly qemu fork, also as OCI image.
- Simple firmware/boot loader. (~4 kiB)
- Some client applications: tkey-ssh-agent, tkey-verification, tkey-sign, tkey-random-generator, and their device applications.
- Client libraries: Go. Python and Java PoC.
- External: TypeScript using WebSerial!
- Device libraries: C supported, from LLVM-15.
- External: Rust (rusTKey).
- Initial bringup of Zig.
- tkey-builder OCI image for podman/docker.

Status

- First official hardware release in March 2023: https://shop.tillitis.se/
- Several client apps available for Linux, macOS, and Windows.
- Reproducible builds:
  - For FPGA bitstream, firmware.
  - For all TKey device apps.
  - For client apps, but not on macOS (shared libs).
- Device verification service up and running.
- Everything released on Github: https://github.com/tillitis/

The emulator

Why an emulator?

- Allows a developer of both firmware and device apps:
  - to inspect,
  - debug,
  - validate functions,
  - observe memory like the stack, registers, …

  while running the real, unchanged, software.

- Allows chip engineer/software engineer
  - Experiment with the hardware/software interface.
  - Better communicate ideas to software developers.
- During security audits and testing:
  - Verify memory locations.
  - Observe internal protocols (coming).

QEMU - the gold standard

- Already had RV32IMC support!
- Bonus: Compatible flash chip support for next release, codename Castor.
- We had some previous experience, but not much.
- Huge, complex.

Goal

- Always develop new hardare/software interface in emulator first.
- Experiment with software on top of interface, get a feeling. Difficult to get feeling from specs.
- Use for development of firmware.
- Use for all development of device and client apps.

Reality

- Worked in the beginning.
- Enormous payoff in explorative development.

- Very good for communication between hardware and software engineers.
- Very good for initial firmware development. Could begin without hardware.

However:

- Sometimes hard to keep up with hardware development. Priority on *other* software development.

Using the emulator

Starting a TKey emulator

```
$ qemu-system-riscv32 -nographic -M tk1,fifo=chrid,htif=on \
-bios qemu_firmware.elf -chardev pty,id=chrid -s -d guest_errors

...

char device redirected to /dev/pts/12 (label chrid)"
```

Interacting with emulated TKey

```
$ tkey-runapp --port /dev/pts/12 app.bin
```

Debugging with GDB and qemu

qemu can speak GDB's remote protocol if you start with `-s` or `-S`.

```
$ riscv32-elf-gdb app.elf \
-ex "set architecture riscv:rv32" \
-ex "target remote :1234"
```

NOTE: You feed the `.bin` to qemu, but you debug the `.elf`!

Staying up to date with upstream

- qemu moves *a lot*!
- Last time we tried to catch up to upstream:

  7,079 files changed +780821 -415571 lines changed

Story time!

- Both pro and con qemu.
- Time-of-check, time-of-use bug in `signer` found by Sergei Volokitin, Hexplot.
- Full writeup: https://bugbounty.tillitis.se/security-bulletins/tillitis-security-bulletin-240115-1/

The bug

- When you sent a message to the `signer` app to be signed, you specified a length.
- If this length was too large, you received an error, but the length you sent **was still used** (time-of-check, time-of-use)!

The exploit

- This can leak memory.
- Exploit: Set increasingly larger size (perhaps 1 extra byte) of the message to sign.
- Send max size (4 k) message.
- Get some already existing RAM included in the signature.
- Try to verify the signature on all 256 values of the extra byte until it verifies.
- Profit!
- Repeat for more bytes…

Fix

- Easy. Just refuse to set the internal size variable if it's wrong.
- Or…?

Investigation

- Find out what we leaked.
- Depends on order things are in memory?

Order in memory

In one of the versions we investigated (from gdb):

```
message = 0x4001ee98
secret_key = 0x4001edf8
r = 0x4001ebe0*
```

Phew!

r is the Ed25519 signing context.

Order in memory on real hardware

- We looked into what happens when we reach the top of the memory on real hardware…
- Hardware wrapped memory access!!!! Aaah!

But the signing context?

- The signing itself also uses the stack!
- Placement matters! Where is it!?
- …before secret key! Phew!

```
-------------------------- <--- 0x40020000
| STACK                    |
| ..                       |
| message ↑               | <--- start of exploit, grows
| ..                       |      towards higher addresses
| Private key              |
| ..                       |
| Signing context          | <--- An exploit stops here
| ..                       |
| - - - - - - - - - - - -  |
| ..                       |
|                          |
| Unused area,             |
| randomized at power up   |
|                          |
| ..                       |
| - - - - - - - - - - - -  |
| APP                      | <--- Wraps and continue to read upwards
--------------------------- <--- 0x40000000
```

The real fix

- Refuse to set size variable if it's too big.
- Guarantee position of private key data compared to message to sign.
- Fix hardware addressing not to wrap RAM access. From tag TK1-24.03. Beginning with serial number 0x01337082 0x00000001.

Future of the emulator

- Emulator kind of thrown together for the things we needed at the time for development.
- Maintained after the fact, not necessarily during development, as we intended.
- Bring emulator up to speed with next release, codename Castor.
- Maybe find something easier to work with, that is more exact and easier to hack on, not that complex, and not with such movement upstreams.
- Several Go implementations of RV32IMC available.
- No flash, though.
- But how hard can it be?

Future of the TKey

The Next Generation - Castor:

- Storage, but isolated per app.
- System calls! “Operating system”.
- Much faster client communication.
- More USB endpoints: HID FIDO, probably CCID.

Summary

- USB stick form factor authenticator.
- …but really a general computer in disguise!
- Which uses measured boot to create unique identities based on what the user **has**, **knows**, and **the software integrity**.
- Developed with our own fork of qemu.
- Emulator helps a lot.
- Verify on real hardware! You might be suprised!

The end

- Michael “MC” Cardell Widerkrantz, **mc@tillitis.se**, **mc@hack.org**
- General inquiries, **hello@tillitis.se**
- `#tillitis @ irc.oftc.org`, `#tillitis:matrix.org`
- Web: https://tillitis.se/
- Buy stuff! https://shop.tillitis.se/
- Developer's handbook: https://dev.tillitis.se/